



Pointer Analysis

CMPUT 497/500

Foundations of Program Analysis

Karim Ali

[@karimhamdanali](https://twitter.com/karimhamdanali)

Previously

- Call graph = calling relationships at compile time
- Sound (no missing edges)
- Precise (few spurious edges)
- On-the-fly call graph construction

Previously

- Andersen-style (e.g., SPARK)
- Rapid Type Analysis (RTA)
 - single points-to set for the program
- Variable Type Analysis (VTA)
 - field-based, simplify SCC, no OTF
- Steensgaard-style
 - equality-based (not subset-based)

Today

other variants of points-to

Points-to Analysis vs Alias Analysis

Points-to Analysis

Points-to Analysis

V

Points-to Analysis

$$\text{points-to}(v) = \{o_1, o_2, \dots\}$$

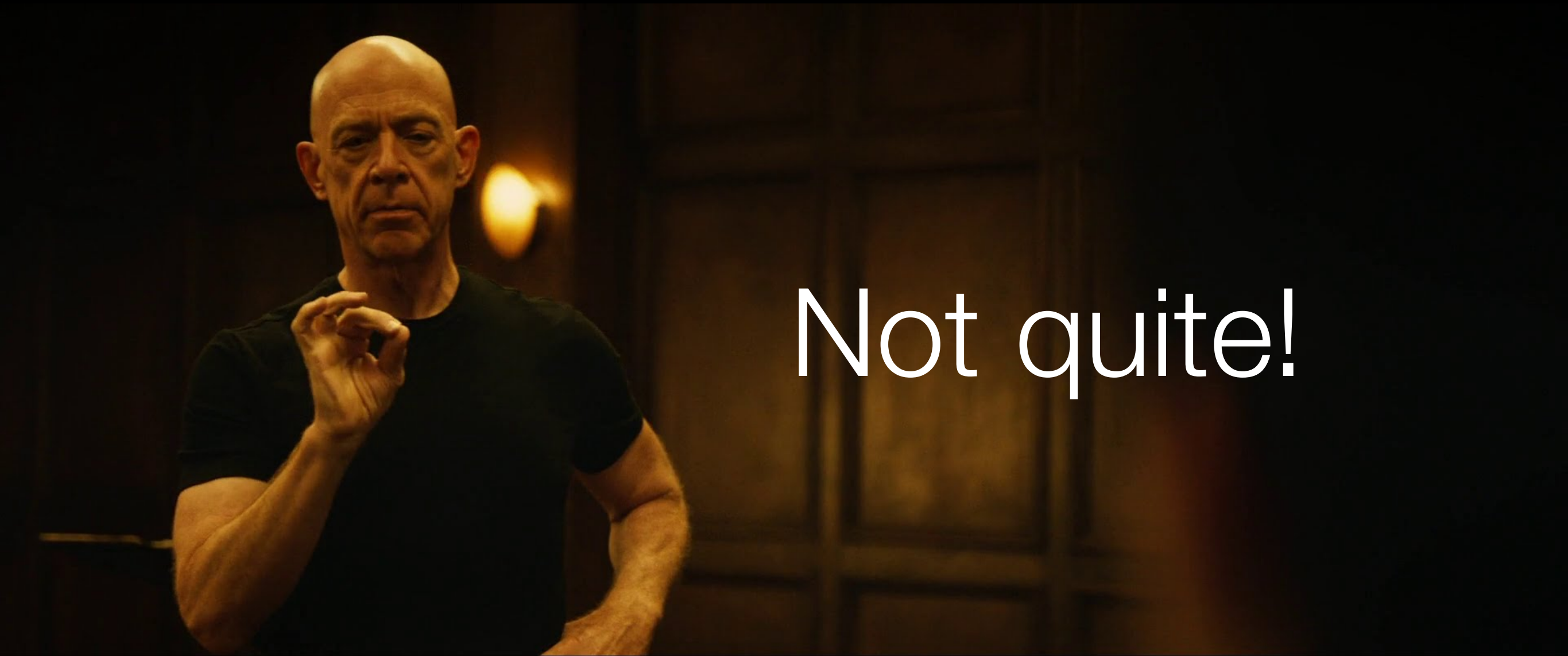
Alias Analysis

Alias Analysis

$V_1 \quad V_2$

Alias Analysis

$\text{alias}(v_1, v_2) = \text{true/false}$



Not quite!

Points-to vs Alias

$$\text{points-to}(v) = \{a_1, a_2, \dots\}$$

The diagram includes three annotations with arrows pointing to parts of the equation:

- An arrow from the text "allocation sites" points to the elements a_1 and a_2 within the set.
- An arrow from the text "may/must" points to the equals sign (=) between the function call and the set.

Points-to vs Alias

$\text{points-to}(v) = \{a_1, a_2, \dots\}$

$\text{may-alias}(v_1, v_2) = \text{true/false}$

$\text{must-alias}(v_1, v_2) = \text{true/false}$

May-Alias vs Must-Alias

```
a = new A();  
if(..) {  
    b = a;  
}  
c = new C();  
d = c;
```

may-alias(a,b) = true

must-alias(a,b) = false

may-alias(a,c) = false

must-alias(c,d) = true

May-Alias vs Must-Alias

```
a = new A();  
if(..) {  
    b = a;  
}  
c = new C();  
d = c;
```

may-alias(a,b) = true

must-alias(a,b) = false

may-alias(a,c) = false

must-alias(c,d) = true

Must-alias is typically associated with control flow!

Must-Alias \Rightarrow Flow-Sensitive?

	<code>b = null;</code>	<code>must-alias(a, s₁, d, s₂) = false</code>
	<code>d = null;</code>	<code>must-alias(a, s₁, d, s₃) = true</code>
<code>s₁:</code>	<code>a = new A();</code>	
	<code>if(..) {</code>	<code>must-alias(b, s₂, c, s₂) = false</code>
	<code>b = a;</code>	<code>must-alias(b, s₂, c, s₃) = false</code>
	<code>}</code>	
<code>s₂:</code>	<code>c = new C();</code>	
	<code>b = c;</code>	
<code>s₃:</code>	<code>d = a;</code>	

Must-Alias \Rightarrow Flow-Sensitive?

	<code>b = null;</code>	<code>must-alias(a,d) =</code>	<code>false</code>
	<code>d = null;</code>	<code>must-alias(a,d) =</code>	<code>true</code>
<code>s₁:</code>	<code>a = new A();</code>		
	<code>if(..) {</code>	<code>must-alias(b,c) =</code>	<code>false</code>
	<code> b = a;</code>	<code>must-alias(b,c) =</code>	<code>false</code>
	<code>}</code>		
<code>s₂:</code>	<code>c = new C();</code>		
	<code>b = c;</code>		
<code>s₃:</code>	<code>d = a;</code>		

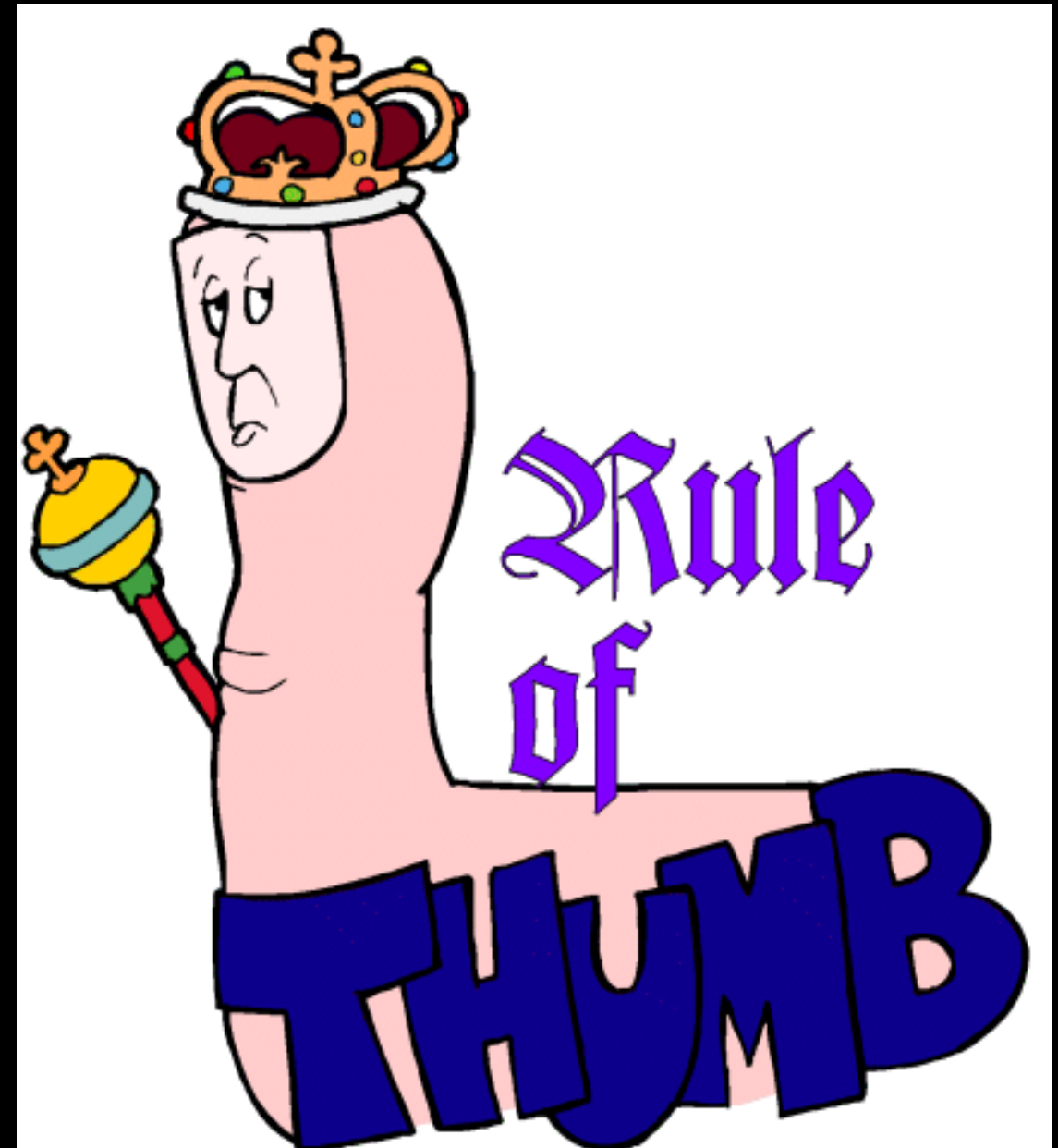
Have to be conservative!

Must-Alias \Rightarrow Flow-Sensitive?

```
    b = null;
    d = null;
s1: a = new A();
    if(..) {
        b = a;
    }
s2: c = new C();
    b = c;
s3: d = a;
```

must-alias(a,d) =	false
must-alias(b,c) =	false

- Must-X analyses must be flow-sensitive.
- May-X analyses may be flow-(in)sensitive.



Points-to as Alias

Points-to as Alias

$$\text{alias}(v_1, v_2) = \text{points-to}(v_1) \cap \text{points-to}(v_2) \neq \emptyset$$

When to use Alias Analysis?

Using Alias Analysis

```
void readProp(String id, String d) {  
    String s = Properties.read(id);  
    if(s==null) s = d;  
    return s;  
}
```

Assume you can't analyze Properties.read()
(e.g., native method, unknown library)

Using Alias Analysis

```
void readProp(String id, String d) {  
    String s = Properties.read(id);  
    if(s==null) s = d;  
    return s;  
}
```

may-alias(s, d) = true

Using Alias Analysis

```
void readProp(String id, String d) {  
    String s = Properties.read(id);  
    if(s==null) s = d;  
    return s;  
}
```

may-alias(s, d) = true

points-to(s) = \emptyset **unsound**

points-to(s) = any object **imprecise**

Using Alias Analysis

```
void readProp(String id, String d) {  
    String s = Properties.read(id);  
    if(s==null) s = d;  
    return s;  
}
```

may-alias(s, d) = true

points-to(s) = ?

points-to(d) = ?

may-alias(s, default) =

points-to(s) \cap points-to(default) $\neq \emptyset$

For Incomplete Programs

- Associating variables with allocation sites is either unsound or imprecise (i.e., points-to)
- Alias analysis is better suited, because it can reason about the relationship between variables without caring about which objects they point to

“Direct” Alias Analysis

“Direct” Alias Analysis

`b = a;`

`c = b;`

`may-alias(b,a) = true`

`may-alias(c,b) = true`

“Direct” Alias Analysis

`a = null;`

`b = a;`

`c = b;`

`may-alias(b,a) = true`

`may-alias(c,b) = true`

*usually
ignored!*

`may-alias(v1,v2) = may-alias-or-both-null(v1,v2)`

When to use Points-to Analysis?

Using Points-to Analysis

```
a1: l = new LinkedList();  
      l.clear();
```

`points-to(l) = { a1 }`

`type-of(points-to(l)) = { LinkedList }`

`l.clear()` can only invoke `LinkedList.clear()`

For method de-virtualization,
alias analysis has almost no use

Weak Updates vs Strong Updates

Weak Updates

- Doable if only *may-alias* info is available
- Retain previous info, and add to it
- Cannot *kill* old info (leads to unsound results)

Weak Updates

- constant propagation
- variables initialized to 0
- only `may-alias(x,y)` is known

$x.f \mapsto 0 \quad y.f \mapsto 0$

$x.f = 3;$

$x.f \mapsto 3 \quad y.f \mapsto 3$

must retain old value of $y.f$

$y.f \mapsto 0$

Strong Updates

- constant propagation
- variables initialized to 0
- `must-alias(x,y)` is known

$x.f \mapsto 0 \quad y.f \mapsto 0$

$x.f = 3;$

$x.f \mapsto 3 \quad y.f \mapsto 3$

can kill old value of $y.f$

$y.f \mapsto 0$

Access Paths

local variable

v.f.g.h...

field accesses

Access Paths as Object Descriptors

`x = new X();`

`{ x }`

`a.f = x;`

`{ x, a.f }`

`b.g = a.f;`

`{ x, a.f, b.g }`

`c.h = b;`

`{ x, a.f, b.g, c.h.g }`

$\{x\}, \{x, y\}$

Encoding Alias Info as Access Paths

```
x = new ..O;  
y = new ..O;  
z = new ..O;
```

{x}, {y}, {z}

if(..)

{x}, {y}, {z}

y = x;

{x,y}, {z}

{x}, {y}, {z}, {x,y}

z = x;

{x,z}, {y}, {x,y,z}

Encoding Alias Info as Access Paths

- $\text{may-alias}(x, y)$ if there is a set containing both x and y
- $\text{must-alias}(x, y)$ if each set that contains x also contains y

$\text{may-alias}(x, y) = \text{true}$

$\text{must-alias}(x, z) = \text{true}$

$\text{must-alias}(x, y) = \text{false}$

$\{x, z\}, \{x, y, z\}$

Strong Updates with Access Paths

Strong Updates with Access Paths

```
x = new ..○;  
y = new ..○;  
z = new ..○;
```

{x}, {y}, {z}

if(..)

{x}, {y}, {z}

y = x;

{x,y}, {z}

{x}, {y}, {z}, {x,y}

z = x;

{x,z}, {y}, {x,y,z}

Strong Updates with Access Paths

- constant propagation
- variables initialized to 0

$\{x\}, \{y\}, \{z\}, \{x,y\}$

$\{x\}.f \mapsto 0, \{y\}.f \mapsto 0,$
 $\{z\}.f \mapsto 0, \{x,y\}.f \mapsto 0$

$z = x;$

$\{x,z\}, \{y\}, \{x,y,z\}$

$\{x,z\}.f \mapsto 0, \{y\}.f \mapsto 0,$
 $\{x,y,z\}.f \mapsto 0$

$x.f = 3;$

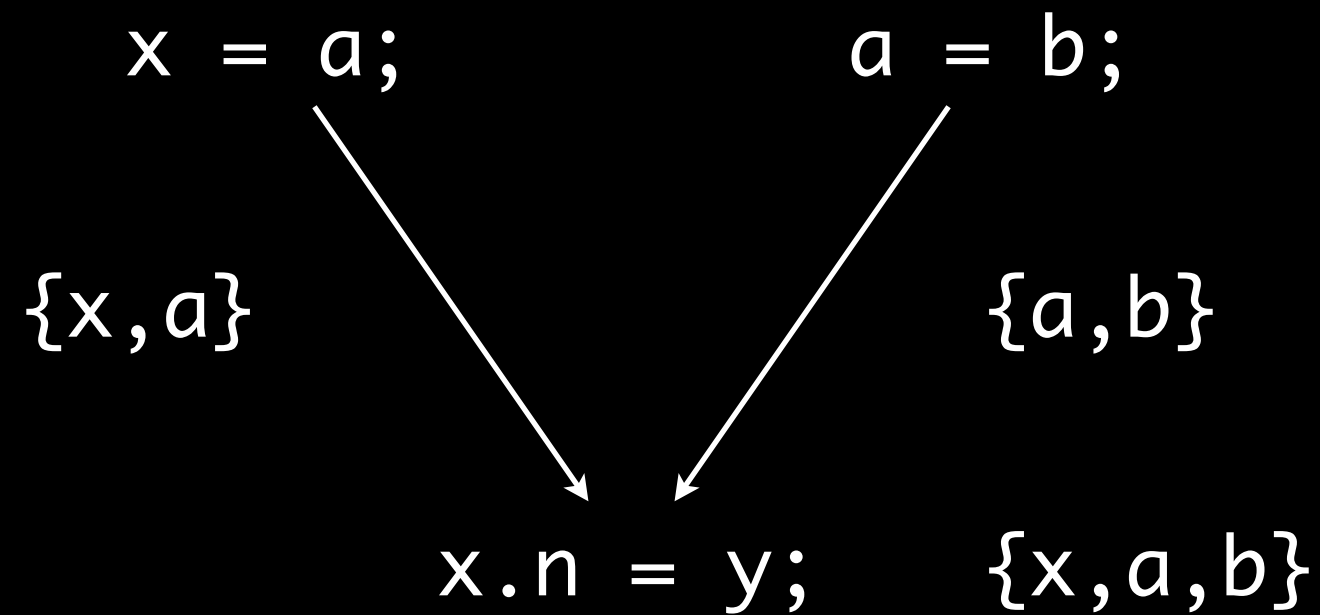
$\{x,z\}, \{y\}, \{x,y,z\}$



Strong
Update

Pointer Analysis & Distributivity

Pointer Analysis & Distributivity



*imprecise
but
distributive*

Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural Pointer Alias Analysis. ACM Transactions on Programming Languages and Systems 21, 4 (July 1999), 848-894.

Pointer Analysis & Distributivity

- In general, pointer analysis is **not** distributive
- Merging first yields different results than merging later
- $f(x \sqcup y) \neq f(x) \sqcup f(y)$

Summary

- Certain Points-to analyses can be used to also answer alias-analysis queries
 - Advantage: re-use points-to analysis results
- Must-alias \Rightarrow flow-sensitive setting
- Strong update requires must-alias information
- Flow-sensitive points-to analysis is not distributive

Next

- Inter-Procedural Analysis